

AD-A262 317



ENTATION PAGE

Form Approved
OPM No

ed to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering information. Send comments regarding this burden estimate or any other aspect of this collection of information, including s Service, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of Management and Budget, Washington, DC 20503

1. AGENCY USE (Leave)		2. REPORT		3. REPORT TYPE AND DATES Final: 30 April 92	
4. TITLE AND Validation Summary Report: TLD Systems, Ltd., TLD VAX/MIL-STD-1750A Ada Compiler System, Version 2.9.0, MicroVAX 3500 under VAX/VMS, Ver. 5.1(Host) to Ver 5.1, TLD MIL-STD-1750A, (Target), 920319W1.11242				5. FUNDING	
6. Wright-Patterson AFB, Dayton, OH USA					
7. PERFORMING ORGANIZATION NAME(S) AND Ada Validation Facility, Language Control Facility ASD/SCEL Bldg. 676, Rm 135 Wright-Patterson AFB, Dayton, OH 45433				8. PERFORMING ORGANIZATION AVF-VSR-528-0392	
9. SPONSORING/MONITORING AGENCY NAME(S) AND Ada Joint Program Office United States Department of Defense Pentagon, Rm 3E114 Washington, D.C. 20301-3081				10. SPONSORING/MONITORING AGENCY	
11. SUPPLEMENTARY					
12a. DISTRIBUTION/AVAILABILITY Approved for public release; distribution unlimited.				12b. DISTRIBUTION	
13. (Maximum 200) TLD Systems, Ltd., TLD VAX/MIL-STD-1750A Ada Compiler System, Version 2.9.0, MicroVAX 3500 under VAX/VMS, Version 5.1 (Host) to MicroVAX 3500, VAX/VMS, Version 5.1 running TLD MIL-STD-1750A Multiple Process Simulator under TLDrtx Real Time Executive, Version 1.0.0 (Target), 920319W1.11242					
<div style="text-align: center;"> </div>					
14. SUBJECT Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANS/MIL-STD-1815A,				15. NUMBER OF	
17. SECURITY CLASSIFICATION UNCLASSIFIED				16. PRICE	
18. SECURITY UNCLASSIFIED		19. SECURITY CLASSIFICATION UNCLASSIFIED		20. LIMITATION OF	

93-06532



6478



8 20 170

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 19 March 1992.

Compiler Name and Version: TLD VAX/MIL-STD-1750A Ada Compiler
System, Version 2.9.0

Host Computer System: MicroVAX 3500,
under VAX/VMS, Version 5.1

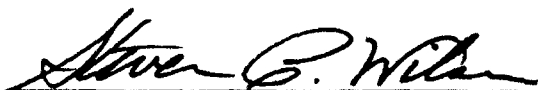
Target Computer System: MicroVAX 3500, VAX/VMS, Version 5.1
running TLD MIL-STD-1750A Multiple
Processor Simulator
under TLDrtx Real Time Executive, Version 1.0.0

Customer Agreement Number: 91-11-14-TLD

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 920319W1.11242 is awarded to TLD Systems, Ltd. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.



Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503



Ada Validation Organization
Director, Computer and Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311



Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Available for Special
A-1	

AVF Control Number: AVF-VSR-528-0392
30 April 1992
91-11-14-TLD

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 920319W1.11242
TLD Systems, Ltd.
TLD VAX/MIL-STD-1750A Ada Compiler System, Version 2.9.0
MicroVAX 3500 under VAX/VMS, Version 5.1 =>
MicroVAX 3500, VAX/VMS, Version 5.1 running
TLD MIL-STD-1750A Multiple Processor Simulator
under TLDrtx Real Time Executive, Version 1.0.0

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

DECLARATION OF CONFORMANCE

Customer: TLD Systems, Ltd.

Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH 45433-6503

ACVC Version: 1.11

Ada Implementation:

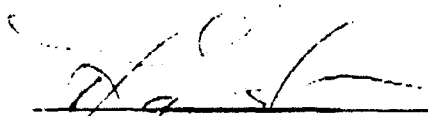
Compiler Name and Version: TLD VAX/MIL-STD-1750A Ada Compiler System,
Version 2.9.0

Host Computer System: MicroVAX 3500, VAX/VMS, Version 5.1

Target Computer System: MicroVAX 3500, VAX/VMS, Version 5.1 running
TLD MIL-STD-1750A Multiple Processor Simulator
TLDrtx Real Time Executive, Version 1.0.0

Customer's Declaration

I, the undersigned, representing TLD Systems, Ltd., declare that TLD Systems, Ltd. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration executing in the default mode. The certificates shall be awarded in TLD Systems, Ltd.'s corporate name.



TLD Systems, Ltd.
Terry L. Dunbar, President

Date: 29 February 1992

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2	REFERENCES.	1-2
1.3	ACVC TEST CLASSES	1-2
1.4	DEFINITION OF TERMS	1-3
CHAPTER 2	IMPLEMENTATION DEPENDENCIES	
2.1	WITHDRAWN TESTS	2-1
2.2	INAPPLICABLE TESTS.	2-1
2.3	TEST MODIFICATIONS.	2-4
CHAPTER 3	PROCESSING INFORMATION	
3.1	TESTING ENVIRONMENT	3-1
3.2	SUMMARY OF TEST RESULTS	3-1
3.3	TEST EXECUTION.	3-2
APPENDIX A	MACRO PARAMETERS	
APPENDIX B	COMPILATION SYSTEM OPTIONS	
APPENDIX C	APPENDIX F OF THE Ada STANDARD	

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

INTRODUCTION

1.2 REFERENCES

- [Ada83] Reference Manual for the Ada Programming Language,
ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
- [Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint
Program Office, August 1990.
- [UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values — for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

INTRODUCTION

Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 2 August 1991.

E28005C	B28006C	C32203A	C34006D	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	B41308B	C43004A
C45114A	C45346A	C45612A	C45612B	C45612C	C45651A
C46022A	B49008A	B49008B	A74006A	C74308A	B83022B
B83022H	B83025B	B83025D	C83026A	B83026B	C83041A
B85001L	C86001F	C94021A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1E02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
AD7206A	BD8002A	BD8004C	CD9005A	CD9005B	CDA201E
CE2107I	CE2117A	CE2117B	CE2119B	CE2205B	CE2405A
CE3111C	CE3116A	CE3118A	CE3411B	CE3412B	CE3607B
CE3607C	CE3607D	CE3812A	CE3814A	CE3902B	

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISC and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

IMPLEMENTATION DEPENDENCIES

The following 285 tests have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113F..Y (20 tests)	C35705F..Y (20 tests)
C35706F..Y (20 tests)	C35707F..Y (20 tests)
C35708F..Y (20 tests)	C35802F..Z (21 tests)
C45241F..Y (20 tests)	C45321F..Y (20 tests)
C45421F..Y (20 tests)	C45521F..Z (21 tests)
C45524F..Z (21 tests)	C45621F..Z (21 tests)
C45641F..Y (20 tests)	C46012F..Z (21 tests)

The following 21 tests check for the predefined type `SHORT_INTEGER`; for this implementation, there is no such type:

C35404B	B36105C	C45231B	C45304B	C45411B
C45412B	C45502B	C45503B	C45504B	C45504E
C45611B	C45613B	C45614B	C45631B	C45632B
B52004E	C55B07B	B55B09D	B86001V	C86006D
CD7101E				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`; for this implementation, there is no such type.

C35713B, C45423B, B86001T, and C86006H check for the predefined type `SHORT_FLOAT`; for this implementation, there is no such type.

C35713D and B86001Z check for a predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`; for this implementation, there is no such type.

A35801E checks that `FLOAT'FIRST..FLOAT'LAST` may be used as a range constraint in a floating-point type declaration; for this implementation, that range exceeds the range of safe numbers of the largest predefined floating-point type and must be rejected. (See section 2.3.)

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a `SYSTEM.MAX_MANTISSA` of 47 or greater; for this implementation, `MAX_MANTISSA` is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain length clauses that specify values for `'SMALL` that are not powers of two or ten; this implementation does not support such values for `'SMALL`.

C45624A..B (2 tests) check that the proper exception is raised if `MACHINE_OVERFLOW` is `FALSE` for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, `MACHINE_OVERFLOW` is `TRUE`.

IMPLEMENTATION DEPENDENCIES

D64005F..G (2 tests) use 10 levels of recursive procedure calls nesting; this level of nesting for procedure calls exceeds the capacity of the compiler.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

LA3004A..B, EA3004C..D, and CA3004E..F (6 tests) check pragma INLINE for procedures and functions; this implementation does not support pragma INLINE.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files (see Section 2.3 regarding CE3413B):

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)
CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)

IMPLEMENTATION DEPENDENCIES

CE3806A..B (2) CE3806D..E (2) CE3806G..H (2) CE3904A..B (2)
CE3905A..C (3) CE3905L CE3906A..C (3) CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 1339 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B44004D B59001E B73004B BA1001A

C34009D and C34009J were graded passed by Evaluation Modification as directed by the AVO. These tests check that 'SIZE for a composite type is greater than or equal to the sum of its components' 'SIZE values; but this issue is addressed by AI-00825, which has not been considered; there is not an obvious interpretation. This implementation represents array components whose length depends on a discriminant with a default value by implicit pointers into the heap space; thus, the 'SIZE of such a record type might be less than the sum of its components 'SIZES, since the size of the heap space that is used by the varying-length array components is not counted as part of the 'SIZE of the record type. These tests were graded passed given that the Report.Result output was "FAILED" and the only Report.Failed output was "INCORRECT 'BASE'SIZE", from line 195 in C34009D and line 193 in C34009J.

A35801E was graded inapplicable by Evaluation Modification as directed by the AVO. The compiler rejects the use of the range FLOAT'FIRST..FLOAT'LAST as the range constraint of a floating-point type declaration because the bounds lie outside of the range of safe numbers (cf. LRM 3.5.7:12).

C83030C and C86007A were graded passed by Test Modification as directed by the AVO. These tests were modified by inserting "PRAGMA ELABORATE (REPORT);" before the package declarations at lines 13 and 11, respectively. Without the pragma, the packages may be elaborated prior to package Report's body, and thus the packages' calls to function REPORT.IDENT INT at lines 14 and 13, respectively, will raise PROGRAM_ERROR.

The tests below were graded passed by Test Modification as directed by the AVO. These tests all use one of the generic support procedures, Length Check or Enum Check (in support files LENCHECK.ADA & ENUMCHEK.ADA), which use the generic procedure Unchecked Conversion. This implementation rejects instantiations of Unchecked Conversion with array types that have non-static index ranges. The AVO ruled that since this issue was not

IMPLEMENTATION DEPENDENCIES

addressed by AI-00590, which addresses required support for Unchecked Conversion, and since AI-00590 is considered not binding under ACVC 1.1I, the support procedures could be modified to remove the use of Unchecked Conversion. Lines 40..43, 50, and 56..58 in LENCHECK and lines 42, 43, and 58..63 in ENUMCHEK were commented out.

CD1009A	CD1009I	CD1009M	CD1009V	CD1009W	CD1C03A
CD1C04D	CD2A21A..C	CD2A22J	CD2A23A..B	CD2A24A	CD2A31A..C
*CD2A81A	CD3014C	CD3014F	CD3015C	CD3015E..F	CD3015H
CD3015K	CD3022A	CD4061A			

*CD2A81A, CD2A81B, CD2A81E, CD2A83A, CD2A83B, CD2A83C, and CD2A83E were graded passed by Test Modification as directed by the AVO. These tests check that operations of an access type are not affected if a 'SIZE clause is given for the type; but the standard customization of the ACVC allows only a single size for access types. This implementation uses a larger size for access types whose designated object is of type STRING. The tests were modified by incrementing the specified size \$ACC_SIZE with '+ 32'.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CE3413B was graded inapplicable by Evaluation Modification as directed by the AVO. This test includes the expression "COUNT'LAST > 150000", which raises CONSTRAINT ERROR on the implicit conversion of the integer literal to type COUNT since COUNT'LAST = 32,767; there is no handler for this exception, so test execution is terminated. The AVO ruled that this behavior was acceptable; the AVO ruled that the test be graded inapplicable because it checks certain file operations and this implementation does not support external files.

Many of the Class A and Class C (executable) test files were combined into single procedures ("bundles") by the AVF, according to information supplied by the customer and guidance from the AVO. This bundling was done in order to reduce the processing time—compiling, linking, and downloading to the target. For each test that was bundled, its context clauses for packages Report and (if present) SYSTEM were commented out, and the modified test was inserted into the declarative part of a block statement in the bundle. The general structure of each bundle was:

IMPLEMENTATION DEPENDENCIES

```
WITH REPORT, TEXT_IO, SYSTEM;
PROCEDURE <BUNDLE_NAME> IS
```

```
-- repeated for each test
```

```
DECLARE
```

```
    <TEST FILE>    [a modified test is inserted here, ...]
```

```
BEGIN
```

```
    <TEST NAME>;    [... and invoked here]
```

```
EXCEPTION      --test is not expected to reach this exception handler
    WHEN OTHERS => REPORT.FAILED("unhandled exception ");
                REPORT.RESULT;
```

```
END;
```

```
TEXT_IO.NEW_LINE;
```

```
--      [... repeated for each test in the bundle]
```

```
TEXT_IO.PUT_LINE ("GROUP TEST <BUNDLE_NAME> COMPLETED");
END <BUNDLE_NAME>;
```

The 1293 tests that were processed in bundles are listed below; each bundle is delimited by '<' and '>'.

<A21001A	A22002A	A22006B	A26004A	A26007A	A27003A	A27004A
A29002A	A29002B	A29002C	A29002D	A29002E	A29002F	A29002G
A29002H	A29002I	A29002J	A29003A	A2A031A>	<A32203B	A32203C
A32203D	A33003A	A34017C	A35101B	A35402A	A35502Q	A35502R
A35710A	A35801A	A35801B	A35801F	A35902C	A38106D	A38106E
A38199A	A39005B	A39005C	A39005D	A39005E	A39005F>	<A39005G
A54B01A	A54B02A	A55B12A	A55B13A	A55B14A	A62006D	A71002A
A71004A	A72001A	A73001I	A73001J	A74105B	A74106A	A74106B
A74106C	A74205E	A74205F>	<A83009A	A83009B	A83041B	A83041C
A83041D	A83A02A	A83A02B	A83A06A	A83A08A	A83C01C	A83C01D
A83C01E	A83C01F	A83C01G	A83C01H	A83C01I	A83C01J	A85007D
A85013B	A87B59A>	<AB7006A	AC1015B	AC3106A	AC3206A	AC3207A>
<AD1A01A	AD1A01B	AD1D01E	AD7001B	AD7005A	AD7101A	AD7101C
AD7102A	AD7103A	AD7103C>	<AD7104A	AD7203B	AD7205B>	<C23001A
C23003A	C23006A	C24002A	C24002B	C24002C	C24003A	C24003B
C24003C	C24106A	C24113A	C24113B	C24113C	C24113D	C24113E>
<C24201A	C24202A	C24202B	C24202C	C24203A	C24203B	C24207A
C24211A	C25001A	C25001B	C25003A	C25004A	C26002B	C26006A>
<C26008A	C27001A	C2A001A	C2A001B	C2A001C	C2A002A	C2A006A
C2A008A	C2A009A	C2A021B>	<C32107A	C32107C	C32108A	C32108B
C32111A	C32111B>	<C32112A	C32112B	C32113A>	<C32114A	C32115A
C32115B>	<C32117A	C34001A	C34001C	C34001D	C34001F	C34002A
C34002C	C34003A	C34003C>	<C34004A	C34004C	C34005A	C34005C>
<C34005D	C34005F	C34005G	C34005I>	<C34005J	C34005L	C34005M
C34005O>	<C34005P	C34005R	C34005S	C34005U	C34006A	C34006F
C34006G	C34006J>	<C34006L	C34007A	C34007D	C34007F	C34007G>
<C34007I	C34007J	C34007M	C34007P>	<C34007R	C34007S>	<C34009A
C34009F	C34009G	C34009L	C34011B	C34012A	C34014A	C34014C>

IMPLEMENTATION DEPENDENCIES

<C34014E	C34014G	C34014H	C34014J	C34014L	C34014N	C34014P
C34014R	C34014T>	<C34014U	C34014W	C34014Y	C34015B	C34016B
C34018A	C35003A	C35003B	C35003D	C35003F	C35102A	C35106A
C35404A	C35404C>	<C35503A	C35503B	C35503C	C35503D	C35503E
C35503F	C35503G	C35503H	C35503K>	<C35503L	C35503O	C35503P
C35504A	C35504B	C35505A	C35505B	C35505C>	<C35505D	C35505E
C35505F	C35507A	C35507B>	<C35507C	C35507E	C35507G	C35507H
C35507I	C35507J>	<C35507K	C35507L>	<C35706A	C35706B	C35706C
C35706D	C35706E>	<C35707A	C35707B	C35707C	C35707D	C35707E
C35708A	C35708B	C35708C	C35708D	C35708E>	<C35711A	C35711B
C35712A	C35712B	C35712C	C35713A	C35713C>	<C35801D	C35802A
C35802B	C35802C	C35802D	C35802E>	<C35902A	C35902B	C35902D
C35904A	C35904B	C35A02A	C35A03A	C35A03B	C35A03C	C35A03D>
<C35A03N	C35A03O	C35A03P>	<C35A03Q	C35A04A	C35A04B	C35A04C>
<C35A04D	C35A04N>	<C35A04O	C35A04P>	<C35A04Q	C35A05A	C35A05D
C35A05N>	<C35A05Q	C35A06A	C35A06B>	<C35A06D	C35A06N	C35A06O>
<C35A06P	C35A06Q	C35A06R	C35A06S	C35A07A	C35A07B	C35A07C>
<C35A07D	C35A07N	C35A07O	C35A07P	C35A07Q	C35A08B	C36003A>
<C36174A	C36180A	C36202A	C36202B	C36202C	C36203A	C36204A
C36204B	C36204C>	<C36205A	C36205B	C36205C	C36205D	C36205E
C36205F	C36205G	C36205H>	<C36205I	C36205J	C36205K	C36301A
C36301B	C36302A	C36303A	C36304A	C36305A>	<C37002A	C37003A
C37003B	C37005A	C37006A	C37007A	C37008A	C37008B>	<C37008C
C37009A	C37010A	C37010B	C37012A	C37102B	C37103A	C37105A
C37107A	C37108B	C37206A	C37207A	C37208A	C37208B	C37209A
C37209B	C37210A>	<C37211A	C37211B	C37211C	C37211D	C37211E
C37213A	C37213B	C37213C	C37213D>	<C37213E	C37213F	C37213G
C37213H>	<C37213J	C37213K	C37213L	C37214A>	<C37215A	C37215B>
<C37215C	C37215D	C37215E	C37215F	C37215G	C37215H	C37216A
C37217A	C37217B	C37217C>	<C37304A	C37305A	C37306A	C37307A
C37309A	C37310A	C37312A	C37402A	C37403A>	<C37404A	C37404B
C37405A	C37409A	C37411A	C38002A	C38002B	C38004A	C38004B
C38005A	C38005B	C38005C	C38006A	C38102A	C38102B	C38102C
C38102D	C38102E	C38104A	C38107A	C38107B>	<C38108A	C38201A
C38202A	C39006A	C39006B	C39006D	C39006E	C39006G	C39007A
C39007B	C39008A	C39008B	C39008C>	<C41101D	C41103A	C41103B
C41104A	C41105A	C41106A	C41107A	C41108A	C41201D	C41203A
C41203B>	<C41204A	C41205A	C41206A	C41207A	C41301A	C41303A
C41303B	C41303C	C41303E	C41303F	C41303G	C41303I	C41303J
C41303K	C41303M	C41303N	C41303O	C41303Q	C41303R	C41303S
C41303U	C41303V	C41303W	C41304A>	<C41304B	C41306A	C41306B
C41306C	C41307A	C41307C	C41307D	C41308A	C41308C	C41308D
C41309A>	<C41320A	C41321A	C41322A	C41323A	C41324A	C41325A
C41326A	C41327A	C41328A>	<C41401A	C41402A	C41403A	C41404A
C42005A	C42006A	C42007A	C42007B>	<C42007C	C42007D	C42007E
C42007F	C42007G	C42007H	C42007I>	<C42007J	C42007K	C43003A
C43004B	C43004C	C43103A	C43103B	C43104A>	<C43105A	C43105B
C43106A	C43107A	C43108A	C43204A	C43204C	C43204E	C43204F>
<C43204G	C43204H	C43204I	C43205A	C43205B	C43205C	C43205D
C43205E	C43205F	C43205G	C43205H	C43205I	C43205J	C43205K
C43206A	C43207A	C43207B	C43207C>	<C43207D	C43208A	C43208B
C43209A	C43210A	C43211A	C43212A	C43212C	C43213A>	<C43214A
C43214B	C43214C	C43214D	C43214E	C43214F	C43215A	C43215B
C43222A>	<C43224A	C44003A	C44003D	C44003E	C44003F	C44003G

IMPLEMENTATION DEPENDENCIES

C45101A	C45101B	C45101C	C45101E	C45101G	C45101H	C45101I
C45101K	C45104A	C45111A	C45111B	C45111C>	<C45111D	C45111E
C45112A	C45112B	C45113A>	<C45114B	C45122A	C45122B	C45122C
C45122D	C45123A	C45123B	C45123C>	<C45201A	C45201B	C45202A
C45202B	C45210A	C45211A	C45220A	C45220B	C45220C	C45220D
C45220E	C45220F	C45231A	C45231C>	<C45232A	C45232B	C45241A
C45241B	C45241C	C45241D	C45241E>	<C45242A	C45242B	C45251A
C45252A	C45252B	C45253A	C45262A>	<C45272A	C45273A	C45274A
C45274B	C45274C	C45281A	C45282A	C45282B	C45291A	C45303A
C45304A	C45304C>	<C45321A	C45321B	C45321C	C45321D	C45321E>
<C45323A	C45331A	C45331D	C45332A	C45342A	C45343A	C45344A
C45345A	C45345B	C45345C	C45345D>	<C45347A	C45347B	C45347C
C45347D	C45411A	C45411C	C45411D	C45412A	C45412C>	<C45413A
C45421A	C45421B	C45421C	C45421D	C45421E>	<C45422A	C45431A
C45502A	C45502C	C45503A	C45503C>	<C45504A	C45504C	C45504D
C45504F>	<C45505A	C45521A	C45521B	C45521C	C45521D	C45521E>
<C45523A	C45524A	C45524B	C45524C	C45524D	C45524E>	<C45532A
C45532B	C45532C	C45532D	C45532E	C45532F	C45532G	C45532H
C45532I	C45532J	C45532K	C45532L>	<C45534A	C45611A	C45611C
C45613A	C45613C	C45614A	C45614C	C45621A	C45621B	C45621C
C45621D	C45621E>	<C45622A	C45624A	C45624B	C45631A	C45631C
C45632A	C45632C	C45641A	C45641B	C45641C	C45641D	C45641E>
<C45652A	C45662A	C45662B	C45672A	C46011A	C46012A	C46012B
C46012C>	<C46012D	C46012E>	<C46013A	C46014A	C46021A	C46023A
C46024A	C46031A	C46032A	C46033A>	<C46041A	C46042A	C46043A
C46043B>	<C46044A	C46044B	C46051A	C46051B	C46051C>	<C46052A
C46053A	C46054A	C47002A	C47002B	C47002C	C47002D	C47003A
C47004A	C47005A	C47006A	C47007A>	<C47008A	C47009A	C47009B
C48004A	C48004B	C48004C	C48004D	C48004E	C48004F	C48005A
C48005B	C48005C	C48006A	C48006B>	<C48007A	C48007B	C48007C
C48008A	C48008B	C48008C	C48008D	C48009A	C48009B	C48009C
C48009D	C48009E	C48009F	C48009G>	<C48009H	C48009I	C48009J
C48010A	C48011A	C48012A	C49020A	C49021A	C49022A	C49022B
C49022C	C49023A	C49024A	C49025A	C49026A>	<C4A005A	C4A005B
C4A006A	C4A007A	C4A010A	C4A010B	C4A010D	C4A011A	C4A012A
C4A012B	C4A013A	C4A013B	C4A014A>	<C51002A	C51004A	C52001A
C52001B	C52001C	C52005A	C52005B	C52005C	C52005D	C52005E
C52005F>	<C52007A	C52008A	C52008B	C52009A	C52009B	C52010A
C52011A	C52011B	C52012A	C52012B	C52013A>	<C52103B	C52103C
C52103F	C52103G	C52103H	C52103K	C52103L>	<C52103M	C52103P
C52103Q	C52103R	C52103S	C52103X	C52104A	C52104B	C52104C
C52104F>	<C52104G	C52104H	C52104K	C52104L	C52104M	C52104P
C52104Q	C52104R	C52104X-	C52104Y>	<C53004B	C53005A	C53005B
C53006A	C53006B	C53007A	C53008A	C54A03A	C54A04A	C54A06A
C54A07A	C54A11A	C54A13A	C54A13B	C54A13C>	<C54A13D	C54A22A
C54A23A	C54A24A	C54A24B	C54A26A	C54A27A	C54A41A	C54A42A
C54A42B	C54A42C	C54A42D	C54A42E	C54A42F	C54A42G	C55B03A
C55B04A	C55B05A	C55B06A	C55B06B	C55B07A>	<C55B08A	C55B09A
C55B10A	C55B11A	C55B11B	C55B15A	C55B16A	C55C01A	C55C02A
C55C02B	C55C03A	C55C03B	C55D01A	C56002A	C57002A	C57003A
C57004A	C57004B	C57004C	C57005A>	<C58004A	C58004B	C58004C
C58004D	C58004F	C58004G	C58005A	C58005B	C58005H	C58006A
C58006B	C59001B	C59002A	C59002B	C59002C>	<C61008A	C61009A
C61010A	C62002A	C62003A	C62003B	C62004A	C62006A	C62009A

IMPLEMENTATION DEPENDENCIES

C63004A	C64002B>	<C64004G	C64005A	C64005B	C64005C	C64103A
C64103B	C64103C	C64103D	C64103E	C64103F>	<C64104A	C64104B
C64104C	C64104D	C64104E	C64104F	C64104G	C64104H	C64104I
C64104J	C64104K	C64104L	C64104M	C64104N	C64104O	C64105A
C64105B	C64105C	C64105D	C64105E	C64105F>	<C64106A	C64106B
C64106C	C64106D	C64107A	C64108A	C64109A	C64109B	C64109C
C64109D	C64109E>	<C64109F	C64109G	C64109H	C64109I	C64109J
C64109K	C64109L>	<C64201B	C64201C	C64202A	C65003A>	<C65003B
C65004A	C66002A	C66002C	C66002D	C66002E	C66002F	C66002G
C67002A	C67002B	C67002C	C67002D	C67002E>	<C67003A	C67003B
C67003C	C67003D	C67003E	C67005A	C67005B	C67005C	C67005D>
<C72001B	C72002A	C73002A	C73007A	C74004A	C74203A	C74206A
C74207B	C74208A	C74208B	C74209A	C74210A	C74211A	C74211B
C74302A	C74302B	C74305A	C74305B	C74306A	C74307A>	<C74401D
C74401E	C74401K	C74401Q	C74402A	C74402B	C74406A	C74407B
C74409B>	<C83007A	C83012D	C83022A	C83023A	C83024A	C83025A>
<C83027A	C83027C	C83028A	C83029A	C83030A>	<C83031A	C83031C
C83031E	C83032A	C83033A	C83051A	C83B02A	C83B02B	C83E02A
C83E02B	C83E03A	C83E04A	C83F01A	C83F03A	C84002A	C84005A
C84008A	C84009A	C85004B	C85005A	C85005B	C85005C	C85005D>
<C85005E	C85005F	C85005G	C85006A>	<C85006F	C85006G>	<C87A05A
C87A05B	C87B02A	C87B02B	C87B03A	C87B04A	C87B04B	C87B04C
C87B05A	C87B06A	C87B07A	C87B07B>	<C87B07C	C87B07D	C87B07E
C87B08A	C87B09A	C87B09B	C87B09C	C87B10A	C87B11A	C87B11B
C87B13A	C87B14A	C87B14B	C87B14C	C87B14D>	<C87B15A	C87B16A
C87B17A	C87B18A	C87B18B	C87B19A	C87B23A	C87B24A>	<C87B33A
C87B34A	C87B34B	C87B34C	C87B35A	C87B35B	C87B35C	C87B37A
C87B37B	C87B37C	C87B37D	C87B37E	C87B37F	C87B38A	C87B39A>
<C87B40A	C87B41A	C87B42A	C87B43A	C87B44A	C87B45A	C87B45C
C87B47A	C87B48A	C87B48B	C87B50A	C87B54A	C87B57A	C87B62A
C87B62B>	<CB1001A	CB1002A	CB1003A	CB1004A	CB1005A	CB1010A
CB1010B	CB1010C	CB1010D>	<CB2004A	CB2005A	CB2006A	CB2007A
CB3003A	CB3003B>	<CB3004A	CB4001A	CB4002A	CB4003A	CB4004A
CB4005A	CB4006A	CB4007A	CB4008A	CB4009A	CB4013A	CB5002A
CB7003A	CB7005A>	<CC1004A	CC1005C	CC1010A>	<CC1010B	CC1018A
CC1104C	CC1107B	CC1111A	CC1204A	CC1207B	CC1220A	CC1221A
CC1221B	CC1221C	CC1221D>	<CC1222A	CC1224A	CC1225A>	<CC1304A
CC1304B	CC1305B	CC1307A	CC1307B	CC1308A	CC1310A>	<CC1311A
CC1311B	CC2002A	CC3004A	CC3007A	CC3011A	CC3011D	CC3012A
CC3015A	CC3106B>	<CC3120A	CC3120B	CC3121A	CC3123A	CC3123B
CC3125A	CC3125B	CC3125C	CC3125D>	<CC3126A	CC3127A	CC3128A
CC3203A	CC3207B	CC3208A	CC3208B>	<CC3208C	CC3220A	CC3221A
CC3222A	CC3223A	CC3224A	CC3225A>	<CC3230A	CC3231A	CC3232A
CC3233A	CC3234A	CC3235A	CC3236A	CC3240A	CC3305A	CC3305B
CC3305C	CC3305D	CC3406A	CC3406B	CC3406C	CC3406D	CC3407A
CC3407B	CC3407C	CC3407D	CC3407E	CC3407F>	<CC3408A	CC3408B
CC3408C	CC3408D	CC3504A	CC3504B	CC3504C	CC3504D	CC3504E
CC3504F>	<CC3504G	CC3504H	CC3504I	CC3504J	CC3504K>	<CC3601A
CC3601C>	<CC3603A	CC3606A	CC3606B	CC3607B>		

* This test listed in two explanations

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report. For technical and sales information about this Ada implementation, contact:

Terry L. Dunbar
TLD Systems, Ltd.
3625 Del Amo Blvd.
Suite 100
Torrance, CA 90503

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system — if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

PROCESSING INFORMATION

a) Total Number of Applicable Tests	3459	
b) Total Number of Withdrawn Tests	95	
c) Processed Inapplicable Tests	67	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	285	
f) Total Number of Inapplicable Tests	616	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The tests were grouped in bundles for more efficient processing. The contents of the magnetic tape were initially loaded on the Sun-4, and moved to the MicroVAX using Ethernet.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the communications link described above, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

<u>Options Switch</u>	<u>Effect</u>
No_Phase	Suppress displaying of phase times during compilation

All tests were executed with the Code Straightening, Global Optimizations, and automatic Inlining options enabled. Where optimizations are detected by the optimizer that represent deletion of test code resulting from unreachable paths, deleteable assignments, or relational tautologies or contradictions, such optimizations are reflected by informational or warning diagnostics in the compilation listings.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	120 -- Value of V
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	'"' & (1..V/2 => 'A') & "'"
\$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & "'"
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

MACRO PARAMETERS

\$MAX_STRING_LITERAL ''' & (1..V-2 => 'A') & '''

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
\$ACC_SIZE	16
\$ALIGNMENT	4
\$COUNT_LAST	511
\$DEFAULT_MEM_SIZE	65536
\$DEFAULT_STOR_UNIT	16
\$DEFAULT_SYS_NAME	AF1750
\$DELTA_DOC	2.0**(-31)
\$ENTRY_ADDRESS	15
\$ENTRY_ADDRESS1	17
\$ENTRY_ADDRESS2	19
\$FIELD_LAST	127
\$FILE_TERMINATOR	ASCII.FS
\$FIXED_NAME	NO_SUCH_FIXED_TYPE
\$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
\$FORM_STRING	""
\$FORM_STRING2	"CANNOT RESTRICT FILE CAPACITY"
\$GREATER_THAN_DURATION	90000.0
\$GREATER_THAN_DURATION BASE LAST	131073.0
\$GREATER_THAN_FLOAT BASE LAST	1.71000E+38
\$GREATER_THAN_FLOAT SAFE LARGE	2.13000E+37

MACRO PARAMETERS

```

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
    NO_SUCH_SHORT_FLOAT_TYPE

$HIGH_PRIORITY        64

$ILLEGAL_EXTERNAL_FILE_NAME1
    BADCHAR@.!

$ILLEGAL_EXTERNAL_FILE_NAME2
    THISFILENAMEWOULDBEPERFECTLYLEGALIFITWERENOTSOLONG.SOTHERE

$INAPPROPRIATE_LINE_LENGTH
    -1

$INAPPROPRIATE_PAGE_LENGTH
    -1

$INCLUDE_PRAGMA1      PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2      PRAGMA INCLUDE ("B28006D1.TST")

$INTEGER_FIRST        -32768
$INTEGER_LAST          32767
$INTEGER_LAST_PLUS_1  32768

$INTERFACE_LANGUAGE   ASSEMBLY

$LESS_THAN_DURATION   -90000.0
$LESS_THAN_DURATION_BASE_FIRST
    -131073.0

$LINE_TERMINATOR      ASCII.CR

$LOW_PRIORITY         1

$MACHINE_CODE_STATEMENT
    R_FMT' (OPCODE=>LR,RA=>R0,RX=>R2);

$MACHINE_CODE_TYPE     ACCUMULATOR

$MANTISSA_DOC          31

$MAX_DIGITS            9

$MAX_INT               2147483647
$MAX_INT_PLUS_1        2147483648

$MIN_INT               -2147483648

$NAME                  NO_SUCH_INTEGER_TYPE

```

MACRO PARAMETERS

\$NAME_LIST	none,ns16000,vax,af1750,z8002, z8001,gould,pdp11,m68000, pe3200,caps,amdahl,i8086, i80286,i80386,z80000,ns32000, ibms1,m68020,nebula,name_x,hp
\$NAME_SPECIFICATION1	NOT_SUPPORTED
\$NAME_SPECIFICATION2	NOT_SUPPORTED
\$NAME_SPECIFICATION3	NOT_SUPPORTED
\$NEG_BASED_INT	16#FFFFFFFE#
\$NEW_MEM_SIZE	65535
\$NEW_STOR_UNIT	16
\$NEW_SYS_NAME	af1750a
\$PAGE_TERMINATOR	ASCII.CR & ASCII.FF
\$RECORD_DEFINITION	RECORD OPCODE: R_OPCODE_VALUE; RA: REGISTER; RX: REGISTER:=R0; END RECORD;
\$RECORD_NAME	R_FMT
\$TASK_SIZE	16
\$TASK_STORAGE_SIZE	2000
\$TICK	1.0/10000.0
\$VARIABLE_ADDRESS	16#8000#
\$VARIABLE_ADDRESS1	16#8020#
\$VARIABLE_ADDRESS2	16#8040#
\$YOUR_PRAGMA	NO_SUCH_PRAGMA

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

Example: With the logical name definition

```
$ DEFINE ADA_DEFAULTS "/LIST/MAC/NOCHECKS"
```

the Ada compiler invocation command

```
$ ADATLD/DEBUG TEST
```

is expanded to

```
$ ADATLD/LIST/MAC/NOCHECKS/DEBUG TEST
```

by the compiler.

3.4 COMPILER OPTION SWITCHES

Compiler option switches provide control over various processing and output features of the compiler. These features include several varieties of listing output, the level and kinds of optimizations desired, the choice of target computer, and the operation of the compiler in a syntax checking mode only.

Keywords are used for selecting various compiler options. The complement keyword, if it exists, is used to disable a compiler option and is formed by prefixing the switch keyword with "NO".

Switches may be abbreviated to the number of characters required to uniquely identify the switch. For example, the switch "CROSSREF" (explained in the list below) may be uniquely identified by the abbreviation "CR" or any longer abbreviation. In the list of switches, on the following pages, the abbreviations are in bold, the optional extra characters are not bolded.

If an option is not specified by the user, a default setting is assumed. All specified compiler options apply to a single invocation of the compiler.

The default setting of a switch and its meaning are defined in the table below. The meaning of the complement form of a switch is normally the negation of the switch. For some switches, the complement meaning is not obvious; these complement switch keywords are listed separately.

In the description of the switches, the target dependent name *target* is used. The value of this symbol is determined by the value of the TARGET switch.

TLD ADA COMPILER
COMPILER USAGE

1750A-ADA-2
3 - 8

Compiler generated file specifications generally conform to host conventions. Thus, any generated filename is the source filename appended with the default file type. The output file name can be completely or partially specified.

SWITCH NAME

MEANING

16BADDR

32BADDR

-- default

The 32BADDR option causes address computations to be performed using 1750A double precision fixed point data words. If 16BADDR is selected, address computations will be performed using single precision fixed point data words ignoring the possibility of a 1750A Fixed Point Overflow Interrupt due to computation of an address greater than 7FFF hex. Applicable to 1750A target only.

ASM (-emulation-file-spec)

NOASM -- default

The ASM switch selects an assembler output file which contains VAX macro references for assembling and emulation of the target on the VAX (1750A target only).

If no emulation-file-spec is specified, the file name is formed from the file name of the input-file-spec with the file type ".MAR". If only the emulation-file name is specified, a file type of ".MAR" is added to form the full file name. If only the file type is specified, the file name of the input-file-spec is used with the specified file type to form the full file name.

The ASM switch overrides an earlier ASSEMBLY or MACRO switch.

CALL TREE

NOCALL_TREE -- default

This switch is used in conjunction with /ELABORATE and /LIST to cause all .CTI files (corresponding to the complete set of object files being linked for this program) to be read in and a closure of all calls in the program to be computed. The results of this analysis is formatted into a subprogram call tree report.

Note: The call tree will be incomplete if any required compilation unit's .CTI files are missing.

CHECKS -- default
CHECKS(=(check_identifier(...)))
NOCHECKS(=(check_identifier(...)))

When the **CHECKS** switch is used, one or more **check_identifiers** are specified and the specified run time checks are enabled. The status of run time checks associated with unmentioned **check_identifiers** is unchanged.

Without any **check_identifiers**, the **NOCHECKS** switch omits all run time checks. If one or more **check_identifiers** are specified, the specified run time checks are omitted. The status of run time checks associated with unmentioned **check_identifiers** is unchanged.

Checks can be eliminated selectively or completely by source statement pragma **Suppress**. Pragma **Suppress** overrides the checks switch.

Check_identifiers are listed below and are described in the LRM, Section 11.7.

ACCESS_CHECK	DISCRIMINANT_CHECK	DIVISION_CHECK
ELABORATION_CHECK	INDEX_CHECK	LENGTH_CHECK
OVERFLOW_CHECK	RANGE_CHECK	STORAGE_CHECK

CONFIG-value
NOCONFIG-value

The **CONFIG** switch provides a conditional compilation (configuration) capability by determining whether or not source text, introduced or bracketed by special comment constructs, is compiled. For a single line:

--/value source-text

where source-text is compiled only if config = value is specified.

For multiple lines: -

--/value line-1

--/value line-2

· ·
· ·
· ·

--/value line-n

where the construct

```
--/value    line-1
```

```
            line-2
```

```
            .
            .
            .
```

```
--/value    line-n
```

is equivalent.

NOTE: The braces ((and)) must appear in the source code; in this instance, they are not meta-characters. All of the text between -- / (value and --- /) value is compiled or skipped, depending upon whether or not the config=value is present.

CROSSREF
NOCROSSREF -- default

This switch generates a cross reference listing that contains names referenced in the source code. The cross reference listing is included in the listing file; therefore, the LIST switch must be selected or CROSSREF has no effect.

CSEG -- default
NOCSEG

This switch indicates that constants are to be allocated in a control section of their own (1750A target only).

CTI (-CTI-file-spec)
NOCTI -- default

This switch generates a CASE tools interface file. The default filename is derived from the object filename, with a .CTI extension. This switch is required to support the Stack Analysis and/or Call Tree switches.

DEBUG -- default
NODEBUG

This switch selects the production of symbolic debug tables in the relocatable object file.

Alternate abbreviation: **DBG**, **NODBG**

DELASSIGN -- default
NODELASSIGN

This switch optimizes code by deleting redundant assignment:.

NOTE: Use of this switch can cause erroneous source programs to execute with unexpected results if references to access objects are made without regard to the interference semantics of Ada.

DIAGNOSTICS -- default
NODIAGNOSTICS

The **DIAGNOSTICS** switch produces a file compatible with Digital's Language Sensitive Editor and the XinoTech Composer. See Digital's documentation for the Language Sensitive Editor for a detailed explanation of this switch.

?
?
?
?

ELABORATOR

This switch selects generation of a setup program that elaborates all compilation units on which the named subprogram depends and then calls the named program. This program will be the main program at link time.

EXCEPTION_INFO
NOEXCEPTION_INFO -- default

This switch generates a string in the relocatable object code that is the full pathname of the file being compiled. It generates extra instructions to identify the location at which an unhandled exception occurred. The **NOEXCEPTION_INFO** switch suppresses the generation of the string and extra instructions. The source statement `pragma Suppress (ALL_CHECKS)` or `pragma Suppress (EXCEPTION_INFORMATION)` suppresses only the extra instructions.

FULL_CALL_TREE -- default

When the FULL_CALL_TREE switch is used, the compiler listing includes representations of every call.

?
?

INDENT-n
INDENT-3 -- default

This switch produces a formatted (indented) source listing. This switch assigns a value to the number of columns used in indentation; the value n can range from 0 to 15.

INDIRECT
NOINDIRECT -- default

If this switch is used, all subprograms declared in the compilation are called with indirect calls. This switch only applies to the 1750A target.

INFO -- default
NOINFO

This switch produces all diagnostic messages. It suppresses the production of information-level diagnostic messages.

INLINE -- default
NOINLINE

This default switch automatically inlines any procedure that is called only once. It may be disabled by adding the noinline option to the command line. Inlining is only implemented for calls that are made within the same compilation unit as the body of the procedure to be expanded.

?
?
?
?
?

INTSL
NOINTSL -- default

This switch intersperses lines of source code with the assembly code generated in the macro listing. This switch is valid only if the LIST, SOURCE, and MACRO switches are selected, and one of the MACRO, ASM, or ASSEMBLY switches is selected. The MACRO switch overrides an earlier ASSEMBLY or ASM switch.

?
?
?
?
?

LIBRARY=Ada-program-library-file-spec
LIBRARY=target.LIB -- default

This switch identifies the file to be used for Ada Program Library. The default value of target in the Ada Program Library file spec is derived from the TARGET switch.

LIST(-listing-file-spec)
NOLIST -- default in interactive mode
LIST -- default for background processes

The **LIST** switch generates a listing file. The default file type is **.LIS**. The listing-file-spec can be optionally specified.

LOG
NOLOG -- default

This switch requests the compiler to write a compiler log, including command line options and the file spec of the Ada source file being compiled, to **SY\$OUTPUT**.

MACRO
NOMACRO -- default

The **MACRO** switch produces an assembly like object code listing appended to the source listing file. The **LIST** switch must be enabled or this switch has no effect.

MAIN_ELAB
NOMAIN_ELAB -- default

The **MAIN_ELAB** switch is used to inform the compiler that the compilation unit being compiled is to be treated as a user-defined elaboration, or setup, program.

Note: The **XTRA** switch is required when **MAIN_ELAB** is to be used.

MAKELIB(-parent-APL-spec)
NOMAKELIB -- default

The **MAKELIB** switch creates a new Ada Program Library (APL) file. **MAKELIB** should be used with caution because it creates a new APL file in the default directory even if another APL file of the same name existed.

The new APL file is created in the default directory with the name **target.LIB** unless the **LIBRARY** switch is used.

If **MAKELIB** is used without a parent, a new library is created with the default RTS specification. This specification is derived from the name **TLD_LIB_target**. See the target dependent compiler sections for further explanations of this name.

MAXERRORS=*n*
MAXERRORS=500 -- default

This switch assigns a value limit to the number of errors forcing job termination. Once this value is exceeded, the compilation is terminated. Information-level diagnostic messages are not included in the count of errors forcing termination. The specified value's range is from 0 to 500.

MODEL= <i>model</i>	-- 1750A target
MODEL-STANDARD	-- default
MODEL-VAMP	
MODEL-IBM_GVSC	-- IBM_GVSC target
MODEL-HWELL_GVSC	-- Honeywell_GVSC target
MODEL-HWELL_GVSC_FPP	-- Honeywell_GVSC target (with floating point processor)
MODEL-RWELL_ECA	-- Rockwell Embedded Compiler architecture
-R11750AB	-- Rockwell International 1750A/B architecture
MODEL-MA31750	-- Marconi 31750 architecture
MODEL-PACE_1750AE	-- PACE 1750AE architecture
MODEL-MS_1750B_II	-- MIL-STD-1750B, Type II
-MS_1750B_III	-- MIL-STD-1750B, Type III
MODEL-MDC281	-- Marconi MDC281

By default, the compiler produces code for the generic or standard target. The model switch allows the user to specify a nonstandard model for the target.

For the 1750A target, MDC281 switch selects the MDC281 (MAS 2nd) implementation of MIL-STD-1750A.

OBJECT(*-object-file-spec*)
OBJECT -- default
NOOBJECT

The OBJECT switch produces a relocatable object file. The default file type is ".OBJ".

OPT -- default
NOOPT

The OPT switch enables global optimization of the compiled code.

PAGE-n
PAGE=60 -- default

The PAGE switch assigns a value to the number of lines per page for listing. The value can range from 10 to 99.

PARM
NOPARM -- default

The PARM switch causes all option switches governing the compilation, including the defaulted option switches, to be included in the listing file. The LIST option must also be selected or PARM will have no effect. User specified switches are preceded in the listing file by a leading asterisk (*).

PHASE -- default
NOPHASE

The NO_PHASE switch suppresses the display of phase names during compilation.

REF_ID_CASE=option

The Ref_Id_Case switch is used to determine how variable names appear in the compiler listing. The options for this switch are:

All_Lower	-- All variable names are in lower case.
All_Underlined	-- All variable names are underlined.
All_Upper	-- All variable names are in upper case.
As_Is	-- All variable names appear as is.
Initial_Caps	-- All variable names have initial caps.
Insert_Underscore	-- All variable names have underscores inserted.

REF_KEY_CASE=option

The Ref_Key_Case switch is used to determine how Ada key words appear in the compiler listing. The options for this switch are:

All_Lower	-- All Ada key words are in lower case.
All_Underlined	-- All Ada key words are underlined.
All_Upper	-- All Ada key words are in upper case.
As_Is	-- All Ada key words appear as is.
Initial_Caps	-- All Ada key words have initial caps.
Insert_Underscore	-- All Ada key words have underscores inserted.

REFORMAT(-reformat-file-spec)
NOREFORMAT -- default

This switch causes TLDada to reformat the source listing in the listing file and, if a reformat-file-spec is present, to generate a reformatted source file. The default type of the new source file is ".RFM".

SOURCE -- default
NOSOURCE

This switch causes the input source program to be included in the listing file. Unless they are suppressed, diagnostic messages are always included in the listing file.

STACK_ANALYSIS
NOSTACK_ANALYSIS -- default

This switch is used with the ELABORATOR switch to cause all CTI files (corresponding to the complete set of object files being linked for this program) to be read in. The subprogram call tree is analyzed to compute stack requirements for the main program and each dependent task.

NOTE: The tree will be incomplete if any required compilation unit's CTI files are missing.

SYNTAX_ONLY
NOSYNTAX_ONLY -- default

This switch performs syntax and semantic checking on the source program. No object file is produced and the MACRO switch is ignored. The Ada Program Library is not updated.

TARGET-1750A -- default
TARGET-VAX -- default

This switch selects the target computer for which code is to be generated for this compilation. "1750A" selects the MIL-STD-1750A Instruction Set Architecture, Notice A. "VAX" selects the VAX architecture operation under VMS.

WARNINGS -- default
NOWARNINGS

The WARNINGS switch outputs warning and higher-level diagnostic messages.

The NOWARNINGS switch suppresses the output of both warning-level and information-level diagnostic messages.

WIDTH=n
WIDTH=110 -- default

This switch sets the number of characters per line (80 to 132) in the listing file.

WRITE_ELAB
NOWRITE_ELAB -- default

The WRITE_ELAB switch is used to obtain an Ada source file which represents the main elaboration "setup" program created by the compiler. The MAIN_ELAB switch may not be used at the same time as the ELAB switch.

XTRA
NOXTRA -- default

This switch is used to access features under development. See the description of this switch in Section 3.9.

4. DIRECTIVE LANGUAGE

TLDlnk is called by a command which may specify options in a form which is host dependent. See Chapter 5 for a description of the command line on a specific host computer. On each host, one of the options is to specify a linker directive file which is host independent. This section describes the directives which may appear in a linker directive file to control a link operation.

4.1 DIRECTIVE FILE

Each line of the Directive File contains up to 132 characters. Tabs are treated as blanks. Blanks are necessary to separate words when no other punctuation would otherwise separate them, but the number of blanks used is insignificant. Any characters after two successive minuses (--) are ignored. A directive ordinarily consists of one line of input, but an opening parenthesis, "(" or "<", which is unmatched on one line causes all following lines to be included in the same directive until the closing parenthesis, ")" or ">", is found, permitting long parenthesized lists. Words may not be divided between lines. Only one directive is allowed per input line. Either upper or lower case may be used; upper and lower case are equivalent. In the following list of directives and components (e.g., directive attributes), the acceptable abbreviation for a directive is in bold and may be used instead of the entire directive or component name. For example, the CSECT directive attribute **WRITEPROTECT** may be entered as "W."

4.2 DIRECTIVES

TLDlnk directives are individually described in this section and appear in alphabetical order. For discussions of related directives, refer to Sections 3.2 -3.8, in Chapter 3.

In the following descriptions, upper case Roman font is used for keywords and lower case italics indicates information provided by the user, e.g., ADDRESS STATES *decimal*.

Characters inside curly () braces are optional, the user may enter or omit them. Rectangular braces with a vertical bar inside represent a choice; [X|Y] indicates that the user may enter X or Y, but not both.

The - symbol is used for a convenient line break. It is not part of the syntax.

In these descriptions, directive switches are shown with "*" as the lead-in character. For VAX or MV hosted systems, the user should replace "*" with "/". For UNIX hosted systems, "*" should be replaced with "-". For example, the "*TRANSIT" switch (in the NODE description) is entered as "/TRANSIT" for VAX systems, or "-TRANSIT" for UNIX systems.

The following words, in lower case italics, are used in the descriptions:

file

This is a host file specification. A file specification must be completely contained on a line.

node

This is the name of a node in the program being linked.

module

This is the name of a module in the program being linked.

symbol

This is the name of an external symbol in the program being linked.

laddress

This is a logical address, in the form (a.)n[I|O]. In the address, a is a hexadecimal digit giving the address state (default 0), n is a hexadecimal number from 0 to FFFF giving the address within the address state, and I or O (upper or lower case) specifies instruction or operand.

paddress

This is a physical address in the form of a hexadecimal number from 0 to FFFFF.

address

This is a logical or physical address.

lpage

This is a logical page number in the form (a.)n(I|O). In the address, a is the address state (default 0), n is a hexadecimal number from 0 to F giving the page number within the address state, and I or O indicates instruction or operand.

ppage

This is a physical page number in the form of a hexadecimal number from 0 to FF.

decimal

This is a decimal number.

Each TLDlnk directive is described below.

ADDRESS STATES *decimal*

This directive declares the number of page registers which the program being linked is expected to use. If the number is 0, TLDlnk assumes that there are no page registers, and memory mapping is not supported.

If this directive is absent, TLDlnk assumes that the program being linked uses 16 Address States.

ASSIGN *lpage,ppage(,number-pages)*

The ASSIGN directive causes TLDlnk to assign the specified logical page(s) to the specified physical page(s). The assignment begins at lpage and ppage and continues with consecutive logical and physical pages until number-pages have been assigned. If number-pages is omitted, the default is 1 page. The ASSIGN directive is required for all physical pages specified in ROM directives if ADDRESS STATES is greater than zero.

CINCLUDE *file((module((csect,...))(-symbol,...),...))(-symbol,...)*

CINCLUDE, the conditional INCLUDE directive, is no longer supported. Use the INCLUDE directive instead.

COLLECT NODE - *node_name* ([ATTR=*attribute*][NAME=*csect_name*])

The COLLECT NODE directive collects control sections by attribute or name and moves them to the specified node. The control sections collected are those between the last NODE directive and the COLLECT directive which have the specified attribute or control section name.

COMMENT - "Text to be put in Load Module"

The COMMENT directive contains text which TLDlnk puts in the load module. TLDlnk precedes the text within quotes by "/;" to distinguish user inserted comments from those inserted by TLDlnk (which begin with "/;"). All comments specified by COMMENT directives are inserted in the load module immediately following the initial comment which is created by TLDlnk.

CONTINUATION

The CONTINUATION directive indicates that the character following the directive is a continuation line mark for the current directive file and all nested directive files. A continuation line mark is used when more information is needed to complete the current line.

The default continuation marks are operating system-specific: the continuation line mark for computer systems running on UNIX is "\", for VAX/VMS it is "-", and the mark for AOS/VS systems is "&." Continuation line mark characters are set for a directive file when the CONTINUATION directive is followed by the appropriate continuation character.

A continuation line mark must be preceded by a space. The mark cannot cross file boundaries. Continuation line marks only affect lines within the same directive file.

For example, a continuation line in UNIX might look like:

```
store 888 = 1,2,3,4,5,6,7,8,9,10,11,12,13,14, \  
15,16,17
```

and is equivalent to:

```
store 888 = 1,2,3,4,5,6,7,8,9,10,11,12,13,14  
store 896 = 15,16,17
```

A continuation line mark may also be used to place a comment, for example:

store 888 = 1,2,3,4,5,6,7,8,9,10,\-- comment here
11,12,13

NOTE: If the continuation character "-" is to be used in other contexts (e.g., using it on the VAX to exclude symbols on an INCLUDE directive), then the CONTINUATION directive must be used to change the default continuation character.

CSECT *module*, *csect*([, *address*|**ALIGN**-*n*])(**ATTR**-*attribute-list*)

The CSECT directive specifies the address or alignment and/or the attributes of a control section. The module and csect name are required to identify the control section uniquely. Either address or alignment, but not both, may be specified. If address is specified, it is given as a single hexadecimal number. TLDlnk interprets the address as physical if ADDRESS STATES is 0; otherwise, TLDlnk interprets the address as an instruction address or operand address according to the control section attribute. The attributes are identified below. In this list, the characters in bold must be entered, the remaining plain text characters are optional. Italics indicates information provided by the user.

WRITEPROTECT Allocate this control section to a page covered by a page register with the write protect bit on.

NOTWRITEPROTECT Allocate this control section to a page covered by a page register with the write protect bit off.

BLOCKPROTECT Allocate this control section to a 1024-word block protected from processor access by a bit in memory-protect RAM.

NOTBLOCKPROTECT Allocate this control section to a 1024-word block with processor access allowed by a bit in memory-protect RAM.

DMAPROTECT Allocate this control section to a 1024-word block protected from DMA access by a bit in memory-protect RAM.

NOTDMAPROTECT Allocate this control section to a 1024-word block with DMA access allowed by a bit in memory-protect RAM.

STARTROM Allocate this control section to a page designated as startup ROM by a ROM linker directive.

RAM_OR_ROM Allocate this control section to a page designated as RAM_OR_ROM by a ROM linker directive, or if there is no such linker directive or not enough room in the ROM, allocate this control section to RAM.

ROM_ONLY Allocate this control section to a page designated as ROM_ONLY by a ROM linker directive.

RAM_ONLY Allocate this control section to a page not designated as ROM by a linker directive.

DEBUG

DEBUG causes the linker to create a file containing symbols and their values for use by the symbolic debugger. The linker puts all external symbols in the symbol file and any local symbols which were included in the Relocatable Object File by the compiler or assembler. If no file-spec is specified, the name of the symbol file is derived as described in the MAP switch. If DEBUG is not specified, the linker does not produce the symbol file.

DEBUG causes a TLD Symbol File (.sym) to be generated when LDMTYPE = LDM or LLM is specified. DEBUG causes the HP Linker Symbol Files (.L) and an Assembler Symbol File (.A) to be generated whenever LDMTYPE-HP is specified.

END

This directive is always required. In a file specified in a USE directive it terminates directive input from that file. In the primary directive file, it terminates directive input to TLDlnk, so that any subsequent input is ignored. After this directive is read, TLDlnk allocates memory and reads the object files to produce the load module.

ENTRY MODULE *(-)*symbol,...

The option to have TLDlnk produce an entry module file is specified in the command line. See Chapter 5 for a description of the command line options.

If the option to produce an entry module file is specified in the command line, then an ENTRY MODULE directive may be used in the directive file to restrict the entry points which are defined in the entry module file. If the ENTRY MODULE directive does not appear, all external symbols defined in the link are defined in the entry module file. If the ENTRY MODULE directive is used, it must precede any NODE directive.

The symbols listed inside angle brackets in this directive are all preceded by a minus sign, or are all not preceded by a minus sign. If the symbols are not preceded by a minus sign, then only the symbols given are defined in the object module. If the symbols are preceded by a minus sign, then all the entry points in the node except the symbols given are defined in the object module.

EXCLUDE *file* *{(module,...)}*

EXCLUDE, a directive which includes a file while excluding selected modules, is no longer supported. Use the INCLUDE directive instead.

```
INCLUDE(*COND)[file((module_symbol_list)) | file((module_list))-
                (<symbol_list>)]
```

```
module_symbol_list ::=
```

```
module ((csect_list))(<symbol_list>)(,module((csect_list))-
                (<symbol_list>))
```

```
module_list ::=
```

```
module [((csect_list))(,module((csect_list)))... -
        | module(,-module)]
```

```
csect_list ::= csect(,csect...) | -csect(,-csect...)
```

```
symbol_list ::= symbol(,symbol...) | -symbol(,-symbol...)
```

BNF notation is used above to express the complicated syntax of the INCLUDE directive.

This directive causes the specified file to be included in the load module. If any module names are listed in parentheses, all the names must be prefixed with minus signs or none of them may have minus signs. If the module names are preceded by a minus sign, then those object modules are excluded from the load module. If the module names are not preceded by a minus sign, then only the named modules are included in the load module. In either case, the order of the module names is not significant. If no modules are listed in parentheses, then the entire file is included.

If module names are listed without minus signs, each module name may be followed by individual control section names in parentheses following the module name. If any control sections are listed, all the control sections must be prefixed by minus signs or none of them may have minus signs. If control section names are preceded by minus signs, those control sections are excluded from the link. If control section names are not preceded by minus signs, only the named control sections are included in the link. If no list of control section names follows a module name, the entire module is included in the link.

If module names are not listed, or if module names are listed without minus signs, individual external symbol definitions may be included or excluded from individual modules or from the entire file by listing the symbol names, optionally prefixed with minus signs, and enclosed in angle brackets (< >). If a symbol list follows a module name (and its optional list of control section names), the specified symbols are

included or excluded from the module. If a symbol list follows the file name (and the optional list of module names), the specified symbols are included or excluded from the entire file. Symbols may be excluded or included in a directive line, but not both. If the *COND qualifier is used, then the specified modules are included in the load module only if they have not already been included.

INDIRECT (*ADDRESS = address)(*ROMADDRESS = address) symbol,...

The INDIRECT directive specifies symbols that are accessed indirectly, through a branch vector. In this vector, symbols are ordered the same way that they are in the symbol list. The ADDRESS qualifier provides the starting address of the transfer vector. This must be a physical address. The ROMADDRESS qualifier specifies the starting address of a copy of the transfer vector in ROM.

A symbol may only appear once in an INDIRECT symbol list. However, multiple definitions of these are permitted in the object code to permit replacement of procedures. When multiple definitions are used, the transfer vector contains a branch to the last procedure encountered, and no diagnostic is issued.

For more information, see the discussion of "Reprogramming" in Section 3.2.8 of this manual.

LDMTYPE=format(,format...)
LDMTYPE=LDM -- default
LDMTYPE=LLM
LDMTYPE=HP

LDMTYPE specifies the format of the load module and symbol file(s) TLDlnk is to produce. Three formats are currently available. If more than one format is specified, the members of the list are separated by commas. See DEBUG for related information.

- o LDM (file extension .LDM), the default, specifies the TLD Load Module Format.
- o LLM (file extension .LLM) specifies a format that is similar to the TLD Load Module Format, but with logical addresses instead of physical addresses. See Section 3.10.
- o HP (file extension .X) specifies the Hewlett-Packard HP64000 Absolute File format.

LET (*MEMORY_TYPE - "memory type name") symbol = value

The LET directive causes the linker to set the specified symbol to the specified value. The effect is as if the symbol had been defined as an EXPORT in an object module. Any external reference to the specified symbol from an object module will be set to the value specified in the LET directive. Optionally, a symbol type can be declared as a specific memory type if MEMORY_TYPE is set - "memory type name."

MAXADR address

This directive gives the maximum physical address the program is expected to use. If the directive is not used, it is assumed that all 0..FFFF (hexadecimal) locations are available, except those reserved by the RESERVE directive. If the linked program extends beyond the specified address, it is linked with a warning.

MEMORY BLOCK PROTECT

This directive announces that the block protect RAM is available on the target processor, permitting hardware memory protection in increments of 1024 words. If this directive is present, values for loading in memory-protect RAM are included in the load module. If this directive is absent, TLDlnk assumes that there is no block protect RAM.

NODE(*{NO)TRANSIT(~Transit_Name)(*STARTROM)
(*ADDRESS=address | *ALIGN=address) node(,node)

The NODE directive declares that all control sections included up to the next NODE or END directive are contained in the same node. All control sections in a node are visible at the same time.

The TRANSIT switch is used to specify transit routine options. The default is TRANSIT. If NOTTRANSIT is used, the insertion of transit routines is suppressed for calls originating from another node to entry points within this node. If Transit_Name is used, the named transit routine is inserted for all calls originating from another node to entry points within this node.

The STARTROM switch indicates that the contents of this node are to be placed in startup ROM. This switch inserts a /V STARTROM record before the contents of this node in the load module.

The ADDRESS or ALIGN switch, but not both, may be used to specify the start address or the alignment of the first module in the node. If ADDRESS STATES is greater than 0, then the address may be an instruction address, or an operand address, or both.

The first node name is the name of this node. It can duplicate the name of any symbol, file, or module, or it can be a new name. The second node name is the name of the parent of this node. When there is no parent (i.e., for a root node) the second node name is omitted. The same name must not be used as the node name in two NODE directives. The parent node must precede its descendant nodes. NODE directives must be ordered such that no node is separated from its parent node by only its sibling nodes and their descendant nodes.

NOLOAD *address, address*

The NOLOAD directive causes code or data within the specified range to be omitted from the load module. This directive may occur repeatedly to specify multiple ranges. If ADDRESS STATES is 0, the addresses must be physical. If ADDRESS STATES is greater than 0, the addresses may be logical or physical.

NOTE: NOLOAD may be used to suppress generation of code or data that is already in ROM but is referenced by new and or replacement code.

RESERVE *address, address*

This directive announces that no relocatable control sections are to be loaded into the specified range of addresses. Absolute control sections are loaded without regard to reserved areas. Addresses beyond the MAXADR address are treated as reserved. This directive may occur repeatedly for multiple reserved ranges.

If ADDRESS STATES is 0, then the addresses must be physical addresses. If ADDRESS STATES is greater than 0, then the addresses may be logical addresses or physical addresses, but all RESERVES with physical addresses must precede the first NODE directive.

ROM(*switch_list*) *address*,*address*

This directive restricts the given range of physical addresses to control sections designated as read-only memory. All other memory is treated as readable and writeable. This directive may occur repeatedly for separate ROM ranges. If ADDRESS STATES is greater than 0, the ASSIGN directive must specify the logical pages which are to be assigned to all ROM physical pages.

The following list identifies switches that are used to specify attributes of control sections which are allocated to ROM. The attributes of O/I and RAM/ROM are checked for all relocatable control sections. The O/I attribute can have two values: Operand or Instruction. The RAM/ROM attribute can have three values: RAM_ONLY, ROM_ONLY, or RAM_OR_ROM.

- | | | |
|------|--|--|
| *S | Restricts this ROM range to control sections with attribute STARTROM. | |
| *IR | Restricts this ROM range to control sections with attributes Instruction and ROM_ONLY. | |
| *IRR | Restricts this ROM range to control sections with attributes Instruction and RAM_OR_ROM. | |
| *OR | Restricts this ROM range to control sections with attributes Operand and ROM_ONLY. | |
| *ORR | Restricts this ROM range to control sections with attributes Operand and RAM_OR_ROM. | |

Switches can be combined and the combined attributes will be selected. In addition, the following are switches that specify combinations of attributes:

- *I Has the same effect as *IR*IRR. Restricts this ROM range to control sections with attributes Instruction and ROM_ONLY or RAM_OR_ROM.
- *O Has the same effect as *OR*ORR. Restricts this ROM range to control sections with attributes Operand and ROM_ONLY or RAM_OR_ROM.
- *R Has the same effect as *IR*OR. Restricts this ROM range to control sections with attribute ROM_ONLY regardless of the O/I attribute value.
- *RR Has the same effect as *IRR*ORR. Restricts this ROM range to control sections with attribute RAM_OR_ROM regardless of the O/I attribute value.

If no switches are specified, control sections with attributes ROM_ONLY or RAM_or_ROM are allocated to ROM.

SEARCH(*REPEAT)(*NODE=Node_Name) file

This directive causes TLDlnk to search the specified file for modules which define currently undefined external references. Any such modules are included just as if they had been specified in an INCLUDE directive. Undefined weak external references do not cause inclusion on a search, but if an external is both weakly and strongly referenced, its defining module is loaded by SEARCH. New external references from modules included from the search file can cause additional modules to be included from the search file, regardless of the order of modules in the search file. For example, if the program references only S, and S references T, and the library contains T followed by S, then both S and T are included from the library.

The REPEAT switch has two effects. First, it causes TLDlnk to search the file when the END directive is encountered instead of when the SEARCH directive is encountered. Second, the set of files which appear in SEARCH*REPEAT directives is searched repeatedly to try to define new undefined external references from any module included from any file in the set. The REPEAT switch allows the use of multiple libraries which have interlibrary references.

The NODE switch causes TLDlnk to insert any modules included as a result of the search in the specified node. If the NODE switch is not used, the modules are inserted in the node in which the SEARCH occurs, or if the REPEAT switch is used, in the last node of the link.

STORE [*symbol* | *address*] = [*symbol* | *value* | SUM](,...)

The STORE directive indicates that TLDlnk is to cause one or more values to be stored in memory when the program being linked is loaded. The starting location at which values are to be stored is specified either as an address or an external symbol. If ADDRESS STATES is 0, then the address must be a physical address. If ADDRESS STATES is greater than 0, then the address may be a logical address or a physical address. Each value to be stored is specified as a hexadecimal number, or as an external symbol, or as the SUM function described below. TLDlnk causes the first value to be stored in the specified address when the program is loaded into memory. If more than one value is given, the succeeding values are stored in consecutive addresses following the specified address. Without a warning, the values stored in memory by the STORE directive overwrite any other values stored at the same locations.

SUM (*startadr*, *endadr*, *result*, *skipadr*)

The purpose of the SUM function is to return a checksum value. The SUM function returns a value equal to *result* minus the sum of all words from *startadr* to *endadr* with the exception of the word at *skipadr*. Any overflows are ignored in taking the sum. The SUM function is intended to be used with the STORE directive to compute a value to be stored at *skipadr*. This is computed the following way:

$$\text{SUM} = \text{result} - \text{sum}(\text{endadr} - \text{startadr}) - (\text{skipadr})$$

If ADDRESS STATES is 0, then all the addresses used in the SUM function must be physical addresses. If ADDRESS STATES is greater than 0, the addresses may be logical or physical addresses.

USE *file*

This directive causes TLDlnk to read directives from the specified file until it encounters an END directive. Upon encountering the END directive in the specified file, TLDlnk returns to the directive following the USE directive.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

.....

type INTEGER is range -32768 .. 32767;

type FLOAT is digits 6 range $-1.0 \times 2.0^{**127}$.. $0.999999 \times 2.0^{**127}$;

type DURATION is delta $2.0^{*(-14)}$ range -86400.0 .. 86400.0;

type LONG_INTEGER is range -2147483648 .. 2147483647;

type LONG_FLOAT is digits 9
range $-1.0 \times 2.0^{**127}$.. $0.999999999 \times 2.0^{**127}$;

.....

end STANDARD;

APPENDIX F OF THE Ada STANDARD

In the customer's Appendix F documentation that constitutes this appendix, some information appears to be inaccurate or incomplete; the AVF offered the customer an opportunity to redress these points, but the customer declined to do so.

The customer declined to provide the AVF with an updated list of all compiler/linker options and the options used specifically for this validation.

On page C-7, the customer states that the subtype priority is integer range 1 .. 16#3EFF#. However the correct range is 1 .. 64.

On page C-8, the customer states that address clauses for task entries (interrupts) are not supported. However, test B91001H contains such an address clause, and this test was passed. Also, in a petition against this test for a Data General implementation, which does not support such address clauses, the customer asserts that all of the 1750A implementations do support them.

On page C-10, the customer states that the range of priority is 0 to 16366. The correct range is 1 to 64.

On page C-11, the customer states that access objects are implemented as 16-bit integers. However, for tests CD2A81A et al. (See section 2.3), the AVF was requested to increase by 32 bits the size used for access objects whose designated objects are strings.

On page C-12, the customer states that the pragma priority is supported with values of 1 to 16366. The correct range is 1 to 64.

The Ada language definition allows for certain machine dependencies in a controlled manner. No machine-dependent syntax or semantic extensions or restrictions are allowed. The only allowed implementation-dependencies correspond to implementation-dependent pragmas and attributes, certain machine-dependent conventions as mentioned in chapter 13 of the MIL-STD-1815A; and certain allowed restrictions on representation clauses.

The full definition of the implementation-dependent characteristics of the TLD VAX/MIL-STD-1750A Ada Compiler System is presented in this Appendix F.

Implementation-Dependent Pragmas

The TLD ACS supports the following implementation dependent pragmas.

Pragma Collect (type_name, attribute);

This pragma tells the compiler to collect all objects of specified type_name and subtypes of type_name into unmapped control sections. An "unmapped control section" is allocated a physical memory location not covered by a page register. Unmapped control sections are accessed from a device by DMA or by IBM GVSC extended instructions.

Pragma Control_Section (USECT, UNMAPPED, object_name {, object_name...});

This pragma specifies data objects that will be put into unmapped control sections. The first two parameters must be USECT and UNMAPPED and the remaining parameters are the names of Ada objects. An "unmapped control section" is allocated a physical memory location not covered by a page register. Unmapped control sections are accessed from a device by DMA or by IBM GVSC extended instructions.

Pragma Contiguous (type_name);
Pragma Contiguous (object_name);

This pragma is used as a query to determine whether the compiler has allocated the specified type of object in a contiguous block of memory words. The compiler will generate a warning message if the allocation is noncontiguous or is undetermined. The allocation probably will be noncontiguous when data structures have dynamically sized components. The allocation probably will be undetermined when unresolved private types are forward type declarations. This pragma provides information to the programmer about the allocation scheme used by the compiler.

Pragma Export (Language_name, Ada_entity_name, {String});

This pragma is a complement to Pragma Interface and instructs the compiler to make the entity named available for reference by a foreign language module. The language name identifies the language in which the foreign module is coded. The only foreign language presently supported is Assembly. Ada and JOVIAL are permitted and presently mean the same as Assembly but the semantics of their use are subject to redefinition by future releases of the compiler. If the optional third parameter is present, the string provides the name by which the entity may be referenced by the foreign module. The contents of this string must conform to the conventions for the indicated foreign language and the linker being used. No check is made by the compiler to insure that these conventions are obeyed.

Only objects having static allocation and subprograms are supported by pragma Export. If the Ada entity named is a subprogram, this pragma must be placed within the declarative region of the named subprogram. If the name is that of an object, the pragma must be placed within the same declarative region and following the object declaration. It is the responsibility of the programmer to insure that the subprogram and object are elaborated before the reference is made.

```
pragma If (Compile_Time_Expression);
pragma Elself (Compile_Time_Expression);
pragma Else;
pragma Endif;
```

These Source directives may be used to enclose conditionally compiled source to enhance program portability and configuration adaptation. These directives may occur at the place that language defined pragmas, statements, or declarations may occur. Source code following these pragmas will be compiled or ignored similar to the semantics of the corresponding Ada statements depending upon whether the Compile_Time_Expression is true or false, respectively. The primary difference between these pragmas and the corresponding Ada Statements are that the pragmas may enclose declarations and other pragmas.

```
Pragma Interrupt_Kind (Entry_Name, Entry_Type {, Duration } );
```

This pragma must appear in the task specification and must appear after the Entry Name is declared. Allowed Entry_Type are Ordinary, Timed, and Conditional. The optional parameter Duration is applicable only to timed entries and is the time to wait for an accept.

For an Ordinary entry, if the accept is not ready, the task is queried.

For a Timed entry, if the accept is not ready, the program waits for the period of time specified by the Duration. If the accept does not become ready in that period, the interrupt is ignored.

For a Conditional entry, if the accept is not ready, the interrupt is ignored.

Pragma Load (literal_string);

This pragma makes the compiler include a foreign object (identified by the literal_string) into the link command.

Pragma Monitor;

Pragma Monitor can eliminate tasking context overhead. The pragma identifies Ada tasks that obey certain restrictions (listed below), allowing efficient invocation by the compiler. With Pragma Monitor, a simple procedure call is used to invoke task entry.

The pragma only applies to tasks that have the following restrictions:

- o Monitor tasks must only be declared in library level non-generic packages
- o Monitor tasks may contain data declarations only within the accept statement.
- o A monitor task consists of an infinite loop containing one select statement.
- o The "when condition" is not allowed in the select alternative of the select statement.
- o The only selective wait alternative allowed in the select statement is the accept alternative.
- o All executable statements of a monitor task must occur within an accept statement.
- o Only one accept statement is allowed for each entry declared in the task specification.

If a task body violates restrictions placed on monitor tasks, it is identified as erroneous and the compilation fails.

Pragma No_Default_Initialization;

Pragma No_Default_Initialization (typename, {, typename ...});

The LRM requires initialization of certain data structures even though no explicit initialization is coded. For example, the LRM requires access_type objects to have an initial value of "NULL." The

APPENDIX F OF THE Ada STANDARD

No_Default_Initialization pragma would prevent this default initialization.

In addition, initialization declared in a type statement is suppressed by this pragma.

The TLD implementation of packed records or records with representation clauses includes default initialization of filler bits, i.e., bits within the allocated size of a variant that are not associated with a record component for the variant. No_Default_Initialization prevents this default initialization.

No Default Initialization must be placed in the declaration region of the package, before any declaration that require elaboration code. The pragma remains in effect until the end of the compilation unit.

Pragma No_Elaboration;

Pragma No_Elaboration is used to prevent the generation of elaboration code for the containing scope. The pragma must be placed in the declaration region of the affected scope before any declaration that would otherwise produce elaboration code.

Pragma No_Elaboration prevents otherwise unnecessary initialization of packages that are initialized by other non-Ada operations. Examples are ROM data and Read Time Kernel initialization. It is used to maintain the TLD Run Time Library (TLDrt1) and is not intended for general use.

Pragma TCB_Extension (value);

This Pragma is used to extend the size of the Task Control Block on the stack. It can be used only within a task specification. The parameter passed to this program must be static and represents the size to be extended in bytes.

Pragma Within_Page (type_name);

Pragma Within_Page (object_name);

This pragma instructs the compiler to allocate the specified object, or each object of the specified type, as a contiguous block of memory words that does not span any page boundaries (a page is 4096 words).

The compiler will generate a warning message if the allocation is noncontiguous or not yet determined. Additionally, the compiler will generate a warning message if the pragma is in a nonstatic declarative region. If an object exceeds 4096 words, it will be allocated with an address at the beginning of a page, but it will span one or more succeeding page boundaries and a warning message will be produced.

Implementation-Dependent AttributesTask_Id

The `Task_Id` attributes is used only with task objects. This TLD-defined attributes returns the actual system address of the task object.

Specification of Package SYSTEM

Package SYSTEM

The following declarations are defined in package system:

```
type operating_system is ( unix, netos, vms, os_x, msdos, bare );
```

```
type name is (none, ns16000, vax, afl750, z8002, z8001, gould,
  pdp11, m68000, pe3200, caps, amdahl, i8086, i80286, i80386,
  z80000, ns32000, ibmsl, m68020, nebula, name_x, hp);
```

```
system_name: constant name := name'target;
```

```
os_name:      constant operating_system := operating_system'system;
```

```
subtype priority is integer range 1..16#3EFF#; -- 1 is default priority.
```

```
subtype interrupt_priority is integer range 16#3FF0#..16#3FFF#;
```

```
pragma put_line ('>', '>', '>', ' ', system_name,
  ' ', '/', ' ', os_name, ' ', '<', '<', '<');
```

```
type address is range 0 .. 65535;
for address'size use 16;
```

```
type unsigned is range 0 .. 65535;
for unsigned'size use 16;
```

```
type long_address is range 0..16#007FFFFFFF#; -- 23 bit physical address
-- for GVSC
```

-- Language Defined Constants

```
storage_unit: constant := 16;
memory_size:  constant := 65536;
min_int:      constant := -2**31;
max_int:      constant := 2**31-1;
max_digits:   constant := 9;
max_mantissa: constant := 31;
fine_delta:   constant := 2.0**(-31);
tick:         constant := 1.0/10 000.0; -- Clock ticks = 100 msecs.
rtc_tps:      constant := 10 000;
min_delay:    constant := rtc_tps * tick; -- Minimum value of ADA delay
address_0:    constant address := 0;      -- Zero address
```

Restrictions on Representation Clauses

Enumeration representation clauses are supported for value ranges of Integer'First to Integer'Last.

Record representation clauses are supported to arrange record components within a record. Record components may not be specified to cross a word boundary unless they are arranged to encompass two or more whole words. A record component of type record that has itself been "rep specificationed" may only be allocated at bit 0. Bits are numbered from left to right with bit 0 indicating the sign bit.

The alignment clause is not supported.

Address clauses are supported for variable objects and designate the virtual address of the object. The TLD Ada Compiler System treats the address specification as a means to access objects allocated by other than Ada means and accordingly does not treat the clause as a request to allocate the object at the indicated address.

Address clauses are not supported for constant objects, packages, tasks, or task entries.

Implementation-Dependent Names

The TLD Ada Compiler System defines no implementation dependent names for compiler generated components.

Interpretations of Expressions in Address Clauses

Address expression values and type Address represent a location in logical memory (in the program's current address state). For objects, the address specifies a location within the 64K word logical operand space. The 'Address attribute applied to a subprogram represents a 16 bit word address within the logical instruction space.

Restrictions on Unchecked Conversion

Conversion of dynamically sized objects are not allowed.

I/O Package Characteristics

The following implementation-defined types are declared in Text_Io.

subtype Count is integer range 0 .. 511;

subtype Field is Integer range 0 .. 127;

Package Standard

The implementation-defined types of package Standard are:

type Integer is range -32_768 .. 32_767;
 type Long_Integer is range -2_147_483_648 .. 2_147_483_647;
 type Float is digits 6 range -1.0*2.0**127 .. 0.999999*2.0**127;
 type Long_Float is digits 9 range -1.0*2.0**127 .. 0.999999999*2.0**127;
 type Duration is delta 2.0*(-14) range -86_400.0 .. 86_400.0;

Other System Dependencies

LRM Chapter 1.

None.

LRM Chapter 2.

Maximum source line length -- 120 characters.

Source line terminator -- Determined by the Editor used.

Maximum name length -- 120 characters.

External representation of name characters.

Maximum String literal -- 120 characters.

LRM Chapter 3.

LRM defined pragmas are recognized and processed as follows:

Controlled -- Has no effect.

Elaborate -- As described in the LRM.

Inline -- Not presently supported.

Interface -- Supported as a means of importing foreign language components into the Ada Program Library. May be applied either to a subprogram declaration as being specially implemented, -- read Interface as Import --, or to an object that has been declared elsewhere. Interface languages supported are System for producing a call obeying the standard calling conventions except that the BEX instruction is used to cause a software interrupt into the kernel

APPENDIX F OF THE Ada STANDARD

supervisor mode; Assembly for calling assembly language routines; and Mil-Std-1750A for defining built in instruction procedures. An optional third parameter is used to define a name other than the name of the Ada subprogram for interfacing with the linker.

List -- As defined in the LRM.

Memory Size -- Has no effect.

Optimize -- Has no effect. Optimization controlled by compiler command option.

Pack -- Has no effect.

Page -- As defined in the LRM.

Priority -- As defined in the LRM. Priority may range from 0 to 16366. Default priority is 1.

Shared -- As defined in the LRM. May be applied to scalar objects only.

Storage Unit -- Has no effect.

Suppress -- As defined in the LRM for suppressing checks; all standard checks may be suppressed individually as well as "Exception Info" and "All Checks". Suppression of Exception Info eliminates data used to provide symbolic debug information in the event of an unhandled exception. The All Checks selection eliminates all checks with a single pragma. In addition to the pragma, the compiler permits control of check suppression by command line option without the necessity of source changes.

System Name -- Has no effect.

Number declarations are not assigned addresses and their names are not permitted as a prefix to the address attribute. (Clarification only).

Objects are allocated by the compiler to occupy one or more 16 bit 1750A words. Only in the presence record representation clauses are objects allocated to less than a word.

Except for access objects, uninitialized objects contain an undefined value. An attempt to reference the value of an uninitialized object is not detected.

The maximum number of enumeration literals of all types is limited only by available symbol table space.

The predefined integer types are:

Integer range -32_768 .. 32_767 and is implemented as a 1750A single

precision fixed point data.
 Long Integer range -2_147_483_648 .. 2_147_483_647 and implemented
 as 1750A double precision data.
 Short Integer is not supported.
 System.Min_Int is -2_147_483_648.
 System.Max_Int is 2_147_483_647.

The predefined real types are:

Float digits 6.
 Long_Float digits 15.
 Short_Float is not presently supported.
 System.Max_Digits is presently 9 and is implemented as 1750A 48-bit
 floating point data.

Fixed point is implemented as 1750A single and double precision data as is
 appropriate for the range and delta.

On the 1750A, index constraints as well as other address values such as
 access types are limited to an unsigned range of 0 .. 65_536 or a signed
 range of -32_768 .. 32_767.

The maximum array size is limited to the size of virtual memory -- 64K
 words.

The maximum String length is the same as for other arrays.

Access objects are implemented as an unsigned 16 bit 1750A integer. The
 access literal Null is implemented as one word of zero on the 1750A.

There is no limit on the number of dimensions of an array type. Array
 types are passed as parameters opposite unconstrained formal parameters
 using a 3 word dope vector illustrated below:

Word address of first element
Low bound value of first dimension
Upper bound value of first dimension

Additional dimension bounds follow immediately for arrays with more than
 one dimension.

LRM Chapter 4.

Machine_Overflows is True for the 1750A.

Pragma Controlled has no effect for the TLD VAX/1750A Compiler since
 garbage collection is never performed.

APPENDIX F OF THE Ada STANDARD

LRM Chapter 5.

The maximum number of statements in an Ada source program is undefined and limited only by the Symbol Table space.

Case statements unless they are quite sparse, are allowed as indexed jump vectors and are, therefore, quite fast.

Loop statements with a for implementation scheme are implemented most efficiently on the 1750A if the range is in reverse and down to zero.

Data declared in block statements on the 1750A is elaborated as part of its containing scope.

LRM Chapter 6.

Arrays, records and task types are passed on the 1750A by reference.

Pragma Inline is not presently supported for subprograms.

LRM Chapter 7.

Package elaboration is performed dynamically permitting a warm restart without the necessity to reload the program.

LRM Chapter 8.

LRM Chapter 9.

Task objects are implemented as access types pointing to a Task Information Block (TIB).

Type Time in package Calendar is declared as a record containing two double precision integer values: the date in days and the real time clock.

Pragma Priority is supported with a value of 1 to 16366.

Pragma Shared is supported for scalar objects.

LRM Chapter 10.

Multiple Ada Program Libraries are supported with each library containing an optional ancestor library. The predefined packages are contained in the TLD standard library, ADA.LIB.

LRM Chapter 11.

Exceptions are implemented by the TLD Ada Compiler System to take advantage of the normal policy in embedded computer system design to reserve 50% of the duty cycle. By executing a small number of instructions in the prologue of a procedure or block containing an exception handler, a branch may be taken, at the occurrence of an exception, directly to a handler

rather than performing the time consuming code of unwinding procedure calls and stack frames. The philosophy taken is that an exception signals an exceptional condition, perhaps a serious one involving recovery or reconfiguration, and that quick response in this situation is more important and worth the small throughput tradeoff in a real time environment.

LRM Chapter 12.

A single generic instance is generated for a generic body. Generic specifications and bodies need not be compiled together nor need a body be compiled prior to the compilation of an instantiation. Because of the single expansion, this implementation of generics tend to be more favorable on the 1750A because of usual space savings achieved. To achieve this tradeoff, the instantiations must by nature be more general and are, therefore, somewhat less efficient timewise.

LRM Chapter 13.

Representation clause support and restrictions are defined above.

A comprehensive Machine_Code package is provided and supported.

Unchecked_Deallocation and Unchecked_Conversion are supported.

The implementation dependent attributes are all supported except 'Storage_Size for an access type.

LRM Chapter 14.

Full file I/O operations are not supported for the 1750A. Text_Io and Low_Level_Io are supported.